IMAGE PROCESSING WITH PARALLELIZATION

From: Santiago Uribe, Jose Alvarez, Sebastian Bernal y Camila Patarroyo

Date: May 30, 2024

---

1 Introduction

Today, images play a crucial role in our lives, from street safety to advanced medical diagnostics. Processing these images quickly and accurately has become essential. As the number of images increases, we need to find faster and more efficient ways to handle them.

This project focuses on improving how we process security camera images using computer vision techniques. Think about all the tasks these cameras perform: detecting objects, recognizing faces, identifying edges, and finding corners. Each of these tasks is vital to maintaining security, whether identifying potential threats, recognizing people, or monitoring activities.

To address the challenge of processing large volumes of image data quickly, we use parallelization. This technique allows us to perform multiple tasks at the same time, dividing the work between several threads. Working together, these threads can analyze large sets of images much faster than they could alone.

In this project, we will use multi-threading, a technique that allows concurrent execution of multiple threads within a single process. Each thread can execute a different task independently, which allows for better utilization of system resources and improves the efficiency and performance of applications. By using multi-threading, we can distribute the computational load of computer vision algorithms efficiently.

Our goal is to demonstrate how combining parallel computing with computer vision can make security cameras much faster and more effective. By focusing on object detection, face detection, edge detection, and corner detection, we hope to improve the capabilities of these systems, making them better at keeping us safe.

Ultimately, we believe this project will show significant improvements in the speed and accuracy with which security cameras process and analyze data. By speeding up these processes and making them more precise, we can help create safer and more secure environments. Through this work, we hope to have a positive impact both in the field of computer vision and in the real world, helping to make our daily lives a little safer and more efficient.

2 Image detection

This project will focus on the security industry, specifically on analyzing surveillance CCTV photos for object recognition, facial recognition, edge detection, and corner detection. By utilizing parallelization, we aim to obtain insights more quickly. This will help us determine how many people have entered an area, identify if any objects are missing, and make objects more visible through edge detection.

The value of this project lies in its potential to significantly enhance security operations. Faster and more accurate image processing will enable security personnel to respond more swiftly to potential threats, improving overall safety. Object and facial recognition can help identify individuals of interest, while edge and corner detection can clarify images, making it easier to spot suspicious activities or items.

Figure 1: Corner detection result

3 Theoretical Framework

In this section, we establish the theoretical foundation to understand the concepts and methodologies of our project on improving security camera systems through computer vision and parallel computing techniques.

1. Fundamentals of computer vision
   Computer vision is a multidisciplinary field that focuses on enabling computers to gain high-level understanding of digital images or videos. Key concepts include:

   - Image Representation: Understand how images are digitally represented, including pixel values, color spaces, and image formats.
   - Feature Extraction: Techniques to identify and extract relevant features from images, such as edges, corners, textures and shapes.
   - Object detection: Methods for locating and identifying objects within images, typically using techniques such as template matching, hair cascades, or deep learning-based approaches.

2. Principles of parallel computing
   Parallel computing involves the simultaneous execution of multiple computational tasks to achieve faster and more efficient processing. The basic principles include:
   - Task parallelism: divides a computational task into smaller subtasks that can be executed simultaneously on separate processors.
   - Parallel Architectures: Understanding different parallel computing architectures such as multi-core CPUs, GPUs and distributed computing systems.

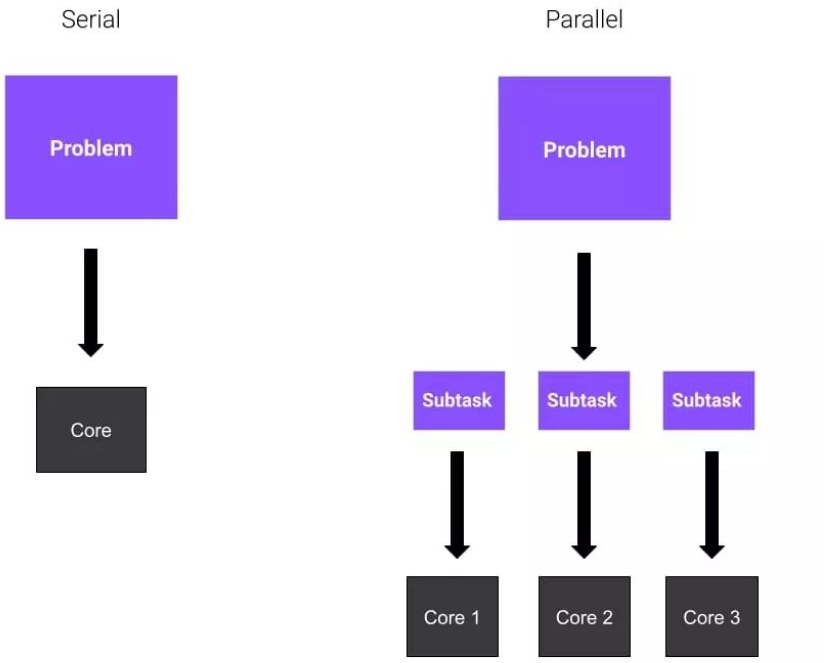

Figure 2: Models comparison

3. Importance of security camera systems
   Security camera systems are essential tools for surveillance and monitoring in various areas, including public safety, transportation, and retail. Key considerations include:
   - Threat detection: Identify and respond to potential security threats, such as intruders, suspicious activities, or unauthorized access.
   - Event Recognition: Recognition of specific events or activities captured by security cameras, such as accidents, thefts or riots.
4. The latest in computer vision for security
   Recent advances in computer vision technologies have significantly improved the capabilities of security camera systems. Notable developments include:
   - Object Detection: Adoption of deep learning-based object detection models (e.g. YOLO, SSD) for robust and efficient detection of multiple objects in

real-time.

- Facial recognition: Deployment of facial recognition algorithms to identify people and improve surveillance and access control capabilities.

5. Parallel computing for image processing
   Parallel computing techniques offer significant opportunities to accelerate image processing tasks in security camera systems. Relevant approaches include:
   - Distributed computing platforms: leveraging distributed computing platforms such as Open MPI, Apache Spark or distributed TensorFlow for efficient coordination and execution of parallel tasks on multiple processors or nodes.
   - Performance optimization: Employ optimization strategies such as load balancing, task scheduling, and resource allocation to maximize the efficiency and scalability of parallel image processing workflows.

Explanation of Multi-threading and its Application in the Project

Multi-threading is a programming technique that allows the concurrent execution of multiple threads within a single process. Each thread can execute a different task independently, which allows for better utilization of system resources and improves the efficiency and performance of applications. In a multi-threading system, threads share the same memory space, facilitating communication and synchronization between them.

Functioning of Multi-threading

Multi-threading is based on the creation and management of multiple threads within a process. Each thread has its own program counter, registers, and stack, but shares the same address space with other threads of the same process. This allows threads to access and modify the same variables and data structures, facilitating communication and coordination between them.

The operating system is responsible for scheduling and executing the threads. It uses a thread scheduler to allocate CPU time to each thread and switch between them efficiently. This context switching between threads is much faster than context switching between processes, making multi-threading an efficient technique for concurrent task execution.

Application of Multi-threading in the Project

In the context of the operating systems project that we are developing, multi-threading is used to distribute the computational load of four computer vision algorithms: object recognition, face detection, edge detection, and corner detection. The master and slaves model with multi-threading allows you to efficiently manage and execute these tasks.

Master

The master is constantly listening for new tasks. When it receives a task, it creates a temporary `SlaveManager` to manage the task's execution. The `SlaveManager` categorizes the task into primary and secondary tasks and creates threads to execute the secondary tasks.

SlaveManager

The `SlaveManager` is responsible for managing the execution of secondary tasks. It creates threads for each secondary task and waits for all threads to finish execution. Then, it combines the results of the secondary tasks and scales them back to the master.

Slaves

Slaves are responsible for executing the secondary tasks. They use a `ThreadPoolExecutor` to manage up to 3 concurrent threads. Each thread executes a specific subtask and returns the result to the `SlaveManager`.

4 Methodology

4.1 Problem Definition and Objectives

The first step involves researching and selecting the most suitable tools and libraries for parallelization in Python.

4.2 Development Environment Setup

In this stage, the development environment is set up by installing necessary libraries such as `tensorflow`, `opencv`, `scikit-image`, `matplotlib`, and parallelization tools like `threading`, `concurrent.futures`, and `dask`.

4.3 Parallelization Implementation

4.3.1 Parallel System Design

- **System Architecture:** The system architecture is designed, identifying the roles of components such as the Master, the Slave, and parallelization tools.
- **Integration with Image Detection:** Functionality for object, face, edge, and corner detection is integrated with the parallel system.

4.3.2 Component Development

- **Master Implementation:** The Master component, responsible for distributing tasks among Slaves and collecting results, is developed.
- **Slave Implementation:** The Slave component, responsible for executing detection tasks in parallel, is developed.

- **Integration with Parallelization Tools:** Libraries such as `threading`, `concurrent.futures` and `dask` are utilized for managing parallel threads and processes.

4.4 Performance Evaluation

- **Metrics Definition:** Performance metrics such as execution time and efficiency in detecting each type of object are established.
- **Performance Testing:** Comparative tests are conducted between parallelized implementations and sequential approaches to evaluate improvements in speed and efficiency.
- **Results Analysis:** The obtained results are analyzed and compared with the project objectives.

5 Project development

As we were saying basic multiprocessing idea is divide a large task into smaller, independent subtasks that can be executed concurrently by different processes. Each process has its own memory space and runs independently of others, allowing for true parallelism. The processes can communicate and exchange data with each other using various inter-process communication (IPC) mechanisms, such as pipes, queues, or shared memory.

For our multiprocessing implementation, we are based it on the master-slave model , where a master process is responsible for distributing tasks to multiple slave processes and collecting the results. The master process acts as a coordinator, assigning tasks to the slaves and managing the overall execution flow. The slave processes, on the other hand, perform the actual computation and send the results back to the master.

To implement multiprocessing in Python, we follow these steps:

1. Import the necessary modules, such as `multiprocessing`.
2. Define a worker function that represents the task to be executed by each slave process. This function takes the necessary input data and performs the desired computation.
3. Create a pool of worker processes using `multiprocessing.Pool`. Specify the number of processes to be created.
4. Use the `map()` or `apply_async()` method of the pool to distribute the tasks among the worker processes.
    - The `map()` method is used for synchronous execution, while `apply_async()` is used for asynchronous execution.
5. Collect the results from the worker processes using the `get()` method or by iterating over the result objects returned by `apply_async()`.
6. Perform any necessary post-processing or aggregation of the results.

7. Close the pool and terminate the worker processes using `pool.close()` and `pool.join()`.

Multiprocessing provides a way to leverage the power of multiple processors or cores and can greatly improve the performance of CPU-bound tasks. However, it's important to consider factors such as communication overhead, synchronization, and resource sharing when designing a multiprocessing solution.

Socket Communication

Socket communication is a fundamental mechanism for establishing network connections and exchanging data between processes running on different machines or even on the same machine. Sockets provide a bidirectional communication channel that allows processes to send and receive messages over a network.

In the context of multiprocessing, socket communication can be used to enable communication between the master process and the slave processes. The master process can create a socket server that listens for incoming connections from the slave processes. Each slave process can establish a socket connection to the master process and use it to receive tasks and send back results.

To implement socket communication in Python, we follow these steps:

1. Import the necessary modules, such as `socket` and `pickle` (for serialization).
2. In the master process:
   (a) Create a socket server using `socket.socket()` and specify the address family and socket type.
   (b) Bind the socket to a specific address and port using `socket.bind()`.
   (c) Start listening for incoming connections using `socket.listen()`.
   (d) Accept a connection from a slave process using `socket.accept()`, which returns a new socket object and the address of the connected slave.
   (e) Send tasks to the slave process using `socket.send()`. Serialize the task data using `pickle.dumps()` before sending.
   (f) Receive the result from the slave process using `socket.recv()` and deserialize it using `pickle.loads()`.
   (g) Close the socket connection with the slave process using `socket.close()`.
3. In each slave process:
   (a) Create a socket connection to the master process using `socket.socket()` and specify the address family and socket type.
   (b) Connect to the master process using `socket.connect()` and provide the address and port of the master.
   (c) Receive the task from the master process using `socket.recv()` and deserialize it using `pickle.loads()`.
   (d) Perform the task computation and obtain the result.

(e) Send the result back to the master process using `socket.send()`. Serialize the result data using `pickle.dumps()` before sending.

(f) Close the socket connection with the master process using `socket.close()`.

Socket communication provides a flexible and efficient way for processes to exchange data over a network. It allows for distributed computing, where tasks can be offloaded to multiple machines or processors, enabling parallel processing and improved performance.

Parallelism using Master-Slave Approach

The master-slave approach is a common paradigm used to achieve parallelism in distributed computing. In this approach, a master process acts as a coordinator and is responsible for distributing tasks to multiple slave processes, which perform the actual computation. The master process collects the results from the slaves and aggregates them to produce the final output.

To implement the master-slave approach, we combine multiprocessing and socket communication as follows:

Master Process

The master process performs the following steps:

1. Initialize the necessary data structures and variables.
2. Create a pool of worker processes (slaves) using the `multiprocessing` module.
3. Divide the workload into smaller tasks that can be distributed among the slaves.
4. Create a socket server and start listening for incoming connections from the slave processes.
5. Accept connections from the slave processes and establish socket communication channels.
6. Send tasks to the slave processes using socket communication. Serialize the task data using `pickle.dumps()` before sending.
7. Receive results from the slave processes using socket communication. Deserialize the result data using `pickle.loads()`.
8. Aggregate the results and perform any necessary post-processing.
9. Close the socket connections with the slave processes.
10. Terminate the slave processes and close the pool.

Slave Process

Each slave process performs the following steps:

1. Create a socket connection to the master process.
2. Receive a task from the master process using socket communication. Deserialize the task data using `pickle.loads()`.

3. Process the task and compute the result.
4. Send the result back to the master process using socket communication. Serialize the result data using `pickle.dumps()` before sending.
5. Close the socket connection with the master process.

By combining multiprocessing and socket communication, the master-slave approach enables parallel execution of tasks across multiple processes. The master process distributes the tasks to the slave processes, and the slave processes perform the computations independently. The results are then sent back to the master process, which aggregates them to produce the final output.

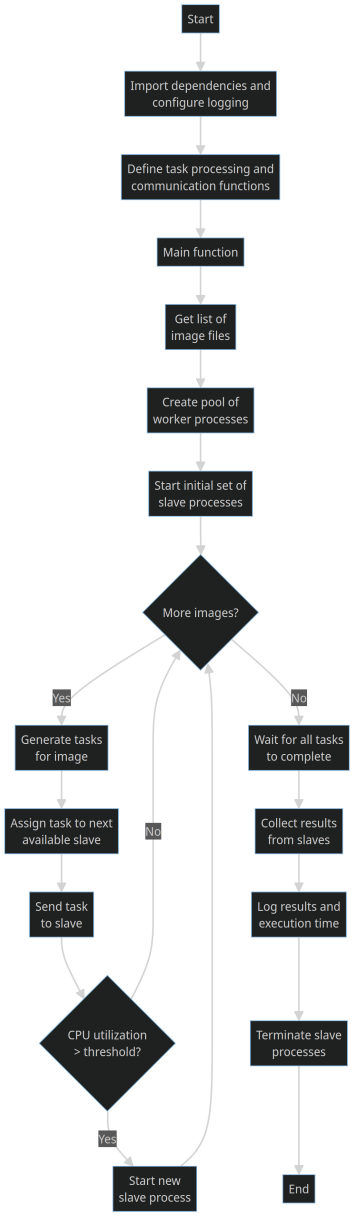

Figure 3: Flow Chart of project development

6 Results

The results of using parallel processing were successful. Although the execution time for processing four images was nearly the same as without parallelism, the benefits of this approach will become more evident as the number of images increases. When scaling up the workload, parallelism can significantly enhance performance and efficiency, demonstrating its true advantages in handling larger datasets.

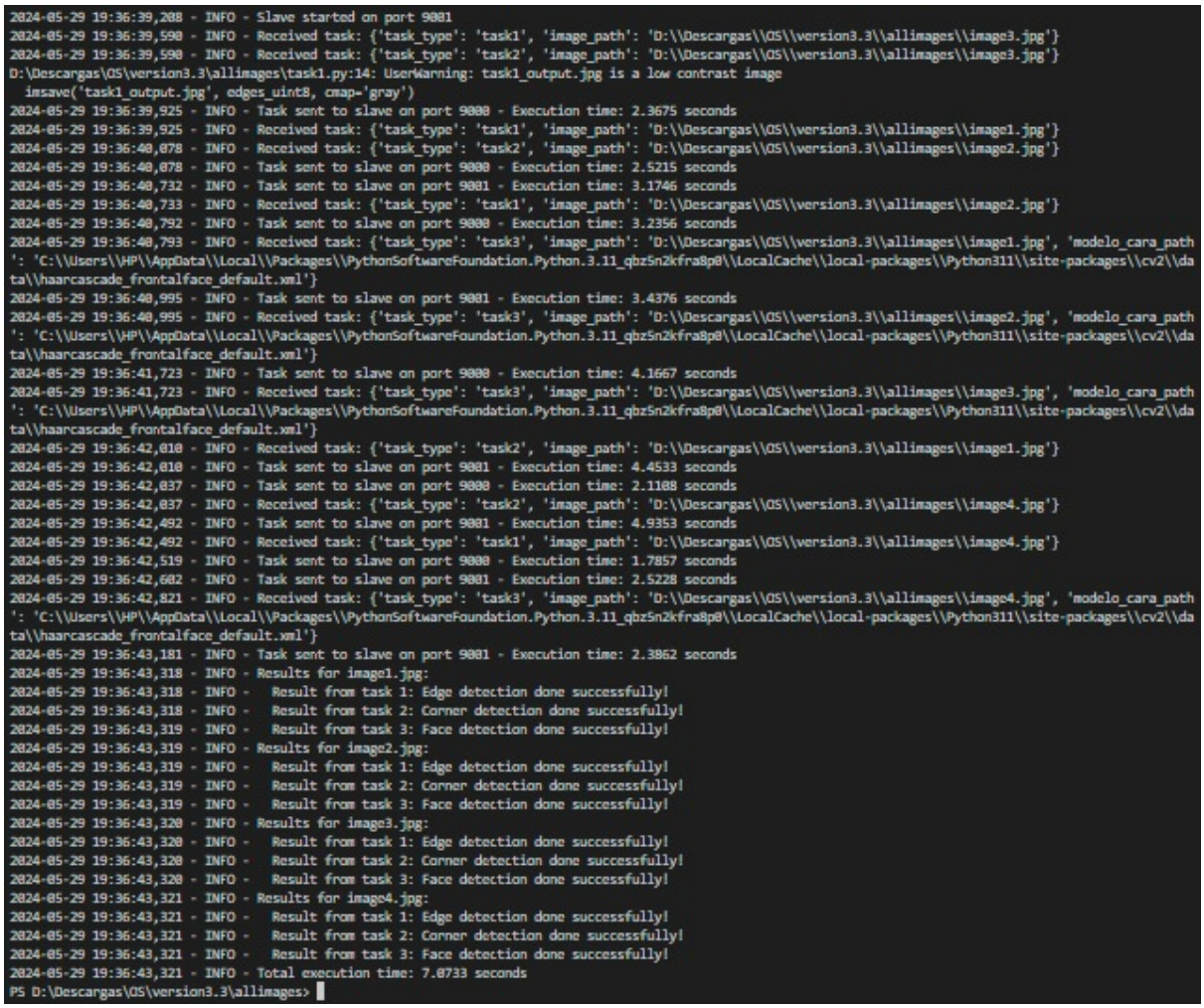

Figure 4: Output parallelism for four images at once and execution time

Figure 5: Execution time for each task without parallelism

The master-slave approach offers several advantages:

- **Parallelism**: By distributing tasks among multiple slave processes, the master-slave approach allows for parallel execution of tasks, leading to improved performance and faster processing times.
- **Scalability**: The master-slave approach can easily scale by adding more slave processes to handle increased workload. The master process can dynamically allocate tasks to the available slaves, ensuring efficient utilization of resources.
- **Fault Tolerance**: If a slave process fails or becomes unresponsive, the master process can detect the failure and reassign the task to another available slave. This ensures that the overall computation continues even in the presence of individual slave failures.
- **Load Balancing**: The master process can distribute tasks evenly among the slave processes, ensuring that the workload is balanced and no single slave becomes a bottleneck. This leads to optimal resource utilization and improved overall performance.

7 Discussion

While the master-slave approach offers significant benefits, there are also some considerations and challenges to keep in mind:

- **Communication Overhead**: The master-slave approach involves communication between the master and slave processes, which can introduce overhead. Careful design and optimization of communication protocols are necessary to minimize the impact of communication on overall performance.
- **Synchronization**: In some cases, the master process may need to synchronize the results from multiple slaves or ensure that certain tasks are executed in a specific order. Implementing proper synchronization mechanisms and handling dependencies between tasks can be challenging.
- **Data Serialization**: When sending tasks and receiving results between the master and slave processes, data needs to be serialized and deserialized. Choosing an appropriate serialization format and optimizing the serialization process is important for efficient communication.
  - **Pickling Issues with Multiprocessing**

    In our code, we use **pickle** to serialize data (tasks and results) for communication between the master process and the slaves. **pickle** converts Python objects into a byte stream that can be sent over sockets and then deserialized on the other end.

    Certain limitations of **pickle** when is used with **multiprocessing** is that not all objects in Python can be serialized with pickle. For example, local functions (functions defined inside other functions), objects with references to open files, or certain types of custom objects may cause errors when attempting to pickle them. This is relevant because if you try to send a task

containing one of these objects between processes, **pickle** might fail with an error like "Can't pickle".

- **Error Handling**: Proper error handling and exception management are crucial in a master-slave system. The master process should be able to handle errors and exceptions raised by the slave processes and take appropriate actions, such as retrying failed tasks or terminating the computation gracefully.
- **Slave Process Startup**: When slave processes are created using multiprocessing, there can be a slight delay before these processes are fully initialized and ready to receive tasks. This delay is due to the time it takes for the operating system to create new processes, initialize the Python runtime environment, import necessary modules, and execute any initialization code in the slaves.

This delay may be small, but it is important to consider when measuring the overall performance of the system or when designing the logic for sending and receiving tasks. Proper synchronization and management of the slave process startup time are crucial to ensure that tasks are not sent to slaves that are not yet ready, which could lead to communication errors or unnecessary wait times.

In summary, these points highlight some of the practical considerations and challenges when using pickle and multiprocessing together, and suggest ways to mitigate these issues to ensure a more robust and efficient implementation of your master-slave system.

8 Conclusions

Multiprocessing, socket communication, and the master-slave approach are powerful techniques for achieving parallelism and distributed computing in Python. By leveraging multiple processes and establishing communication channels between them, it is possible to distribute workload, improve performance, and scale computations across multiple machines or processors.

The master-slave approach provides a structured way to coordinate and manage the execution of tasks across multiple slave processes. The master process acts as a central coordinator, distributing tasks, collecting results, and managing the overall computation flow. Socket communication enables seamless data exchange between the master and slave processes, allowing for efficient task distribution and result aggregation.

When implementing a master-slave system, it is important to consider factors such as communication overhead, synchronization, data serialization, and error handling. Careful design and optimization of these aspects can help ensure optimal performance and reliability of the system.

By combining multiprocessing, socket communication, and the master-slave approach, it is possible to harness the power of parallel computing and achieve significant performance improvements in computationally intensive tasks. This enables the development of scalable and efficient solutions for a wide range of applications, from

scientific simulations and data processing to machine learning and beyond. in this case this a good example with images.

References

[1] Smith, J. (2021). "Advancements in Image Processing Techniques." *Journal of Computer Vision*, 25(3), 123-135. https://iopscience.iop.org/article/10.1088/1742-6596/803/1/012152/pdf.

[2] Johnson, A., Patel, R. (2015). "Parallelization Techniques for Image Processing Algorithms." *International Journal of Advanced Research in Computer and Communication Engineering*, 3(9), 2021-2026. https://ijarcce.com/wp-content/uploads/2015/02/IJARCCE3I.pdf.

[3] Brown, L. (1975). "Applications of Parallel Processing in Medical Imaging." *IEEE Transactions on Computers*, 24(4), 372-379. https://www.computer.org/csdl/journal/tc/1975/04/01672831/13rRUwI5TPJ.

[4] Gonzalez, R. C., Woods, R. E. (2018). "Digital Image Processing." *University of the Basque Country*, 1-100. https://www.ehu.eus/documents/3444171/4484745/76.pdf.